Gas Turbine Design Analysis Functions

In [ ]:

```python
import numpy
import numpy as np
import math
import pandas as pd
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

mechanical_eff = 0.99
gamma_air = 1.4
gamma_g = 1.33333
c_p_air = 1.005
c_p_gas = 1.148
R = 0.287

m_cool_vane_hpt = 0.1450776
m_cool_disc_hpt = 0.0797

# TURBINE INLET
T_01 = 1245.3208220773054        # [k]
P_01 = 1182.073662         # [kPa]
m_dot_1 = 4.835922
M_1 = 0.125

T_02_cycle = 1225.9485919279928

# BETWEEN STATOR AND ROTOR
m_dot_2 = m_dot_1 + m_cool_vane_hpt

# TURBINE EXIT
T_03 = 1041.2535775588708
T_03_cooled = 1033.9843383376274
P_03 = 517.9223058003336
m_dot_3 = m_dot_2 + m_cool_disc_hpt


# BLADE PARAMETERS
# VANE
AR_vane = 0.5
TE_vane = 1.27/1000     # minimum trailing edge thickness in [m]
LE_diameter_vane = 0.000508

# BLADE
AR_rotor = 1.3
TE_rotor = 0.762/1000   # minimum trailing edge thickness in [m]
tip_clearance = 2.0    # minimum tip clearance span -> maximum is 0.02
LE_diameter_rotor = 0.00508             #0.000254*12

"""
This file contains all the main input and functions
"""

class aeroturbine():

    def calc_properties(M, T_stagnation, P_stagnation):
        T = T_stagnation/(1 + ((gamma_g - 1)/2) * M**2)
        P = P_stagnation/((1 + ((gamma_g - 1)/2) * M**2))**(gamma_g/(gamma_g - 1))
        rho = P/(0.287*T)
        c = M * numpy.sqrt(gamma_g * 287 * T) #--

        return T, P, rho, c

    def calc_U(psi):
        """
        This function calculates the metal speed U.
        Input:  Stage loading coefficient "psi"
        Output: Returns the value of U
        """
        U = numpy.sqrt((2*c_p_gas*1000*(T_02_cycle - T_03)) / (psi))
        return U

    def calc_stage_3(U, C_3, T_3, rho_3,P_3,alpha_3):

        #V_w_3 = c_p_gas*1000*(T_01-T_03)/(2*U) + reaction*U
        C_a_3 = C_3 * np.cos(np.radians(alpha_3))
        C_w_3 = math.sqrt(C_3**2 - C_a_3**2)
        V_w_3 = U + C_w_3
        alpha_3 = numpy.rad2deg(numpy.arcsin(C_w_3/C_3))
        V_3 = np.sqrt(V_w_3**2 + C_a_3**2)
        flow_coefficient_3 =  C_a_3 / U
        beta_3 = np.rad2deg(np.arctan(V_w_3 / C_a_3))
        a_3 = np.sqrt(gamma_g * R * 1000 * T_3)
        M_3_rel = V_3 / a_3
        A_3 = m_dot_3/(rho_3 * C_a_3)
```

```python
        P_03_rel = P_3*(1+ (gamma_g-1)/2 * M_3_rel**2)**(gamma_g/(gamma_g-1))

        return C_a_3, C_w_3,  V_3, V_w_3, flow_coefficient_3, beta_3, a_3, M_3_rel, A_3, P_03_rel

    def calc_stage_2(U, reaction, T_1, T_3, P_3, A_3, V_w_3):
        T_2 = T_3 + reaction * (T_1 - T_3)
        P_2 = P_3 * ((T_2/T_3) ** (gamma_g/(gamma_g - 1)))
        rho_2 = P_2 / (R * T_2)
        A_2 = A_3
        C_a_2 = (m_dot_2)/(rho_2 * A_2)
        flow_coefficient_2 = C_a_2 / U
        a_2 = np.sqrt(gamma_g * R * 1000 * T_2)
        # V_w_2 = V_w_3 - (2*reaction * U)
        V_w_2 = (c_p_gas * 1000 * (T_01 - T_03) / (U)) - V_w_3 #This is the connection to work -> we should change this to T_02 maybe
        beta_2 = np.rad2deg(np.arctan(V_w_2 / C_a_2))
        V_2 = np.sqrt(V_w_2**2 + C_a_2**2)
        C_w_2 = (V_w_2 + U)
        C_2 = np.sqrt(C_w_2**2 + C_a_2**2)
        alpha_2 = np.rad2deg(np.arctan(C_w_2/C_a_2))
        M_2 = C_2 / a_2
        M_2_rel = V_2 / a_2
        P_02 = P_2*(1+ (gamma_g-1)/2 * M_2**2)**(gamma_g/(gamma_g-1))
        P_02_rel = P_2*(1+ (gamma_g-1)/2 * M_2_rel**2)**(gamma_g/(gamma_g-1))
        T_02 = T_2 + C_2**2/(2*1000*c_p_gas)



        return T_02, T_2, P_2, rho_2, A_2, C_a_2, flow_coefficient_2, a_2, V_w_2, beta_2, V_2, C_w_2, C_2, alpha_2, M_2, M_2_rel,P_02,P_02_

    def calc_stage_2_trial(U,T_1,T_3,P_3,A_3,V_w_3):

        error = 1
        max_iterations = 10000
        count = 0
        T_2 = 1060
        increment = 0.01
        V_w_2 = (c_p_gas * 1000 * (T_02_cycle - T_03) / (U)) - V_w_3
        C_w_2 = (V_w_2 + U)

        while count < max_iterations:

            P_2 = P_01 * ((T_2/T_02_cycle) ** (gamma_g/(gamma_g - 1))) #changed P_2 = P_3 * ((T_2/T_3)**(gamma_g/(gamma_g-1)))
            rho_2 = P_2/(R*T_2)
            T_02 = T_2 + (C_w_2**2 + (m_dot_2/(rho_2*A_3))**2)/(2*c_p_gas*1000)
            #print(T_02)
            error = np.abs(T_02_cycle - T_02) #cycle_Calc T_02 to be defined as global constant
            count = count + 1
            if (0 < error < 0.001):
                break
            else:
                T_2 = T_2 + increment

        if count != max_iterations:
            reaction = (T_2 - T_3)/(T_1 - T_3)
            A_2 = A_3
            C_a_2 = (m_dot_2)/(rho_2 * A_2)

            flow_coefficient_2 = C_a_2 / U
            a_2 = np.sqrt(gamma_g * R * 1000 * T_2)

            beta_2 = np.rad2deg(np.arctan(V_w_2 / C_a_2))
            V_2 = np.sqrt(V_w_2**2 + C_a_2**2)

            C_2 = np.sqrt(C_w_2**2 + C_a_2**2)
            alpha_2 = np.rad2deg(np.arctan(C_w_2/C_a_2))
            M_2 = C_2 / a_2
            M_2_rel = V_2 / a_2
            P_02 = P_2*(1+ (gamma_g-1)/2 * M_2**2)**(gamma_g/(gamma_g-1))
            P_02_rel = P_2*(1+ (gamma_g-1)/2 * M_2_rel**2)**(gamma_g/(gamma_g-1))

        else:
            T_02 = 0
            T_2 = 0
            P_2 = 0
            rho_2 = 0
            A_2 = 0
            C_a_2 = 0
            flow_coefficient_2 = 0
            a_2 = 0
            V_w_2 = 0
            beta_2 = 0
            V_2 = 0
            C_w_2 = 0
            C_2 = 0
            alpha_2 = 0
            M_2 = 0
            M_2_rel = 0
            P_02 = 0
            P_02_rel = 0
            reaction = 0
```

```python
        return T_02, T_2, P_2, rho_2, A_2, C_a_2, flow_coefficient_2, a_2, V_w_2, beta_2, V_2, C_w_2, C_2, alpha_2, M_2, M_2_rel,P_02,P_02_

    def calc_hub_angles(r_m_pointer, r_hub_pointer, alpha_2_pointer, alpha_3_pointer, flow_coeff_2_pointer, flow_coeff_3_pointer, U_pointer
        alpha_2_hub_rad = numpy.arctan((r_m_pointer/r_hub_pointer) *numpy.tan(numpy.deg2rad(alpha_2_pointer)))
        alpha_2_hub_deg = numpy.rad2deg(alpha_2_hub_rad)

        alpha_3_hub_rad = numpy.arctan((r_m_pointer/r_hub_pointer) *numpy.tan(numpy.deg2rad(alpha_3_pointer)))
        alpha_3_hub_deg = numpy.rad2deg(alpha_3_hub_rad)

        beta_2_hub_rad = numpy.arctan((r_m_pointer/r_hub_pointer) *numpy.tan(numpy.deg2rad(alpha_2_pointer)) - (r_hub_pointer/r_m_pointer)*
        beta_2_hub_deg = numpy.rad2deg(beta_2_hub_rad)

        beta_3_hub_rad = numpy.arctan((r_m_pointer/r_hub_pointer) *numpy.tan(numpy.deg2rad(alpha_3_pointer)) + (r_hub_pointer/r_m_pointer)*
        beta_3_hub_deg = numpy.rad2deg(beta_3_hub_rad)

        U_hub = U_pointer * (r_hub_pointer / r_m_pointer)

        V_2_hub = C_a_2/np.cos(beta_2_hub_rad)
        C_2_hub = C_a_2/np.cos(alpha_2_hub_rad)
        C_3_hub = C_a_3/np.cos(alpha_3_hub_rad)

        T_2_hub = T_02 - (C_2_hub**2)/(2*c_p_gas*1000)
        T_3_hub = T_03 - (C_3_hub**2)/(2*c_p_gas*1000)
        T_1_hub = T_1 #assumed since no free vortexing

        a_2_hub = np.sqrt(gamma_g*T_2_hub*R*1000)

        M_2_rel_hub = V_2_hub / a_2_hub
        M_2_hub = C_2_hub / a_2_hub

        reaction_hub = (T_2_hub-T_3_hub)/(T_1_hub-T_3_hub)

        return alpha_2_hub_deg, alpha_3_hub_deg, beta_2_hub_deg, beta_3_hub_deg,  U_hub, V_2_hub, C_2_hub, M_2_rel_hub, M_2_hub,reaction_hu



    def calc_tip_angles(r_m_pointer, r_tip_pointer, alpha_2_pointer, alpha_3_pointer, flow_coeff_2_pointer, flow_coeff_3_pointer, U_pointer
        alpha_2_tip_rad = numpy.arctan((r_m_pointer/r_tip_pointer) *numpy.tan(numpy.deg2rad(alpha_2_pointer)))
        alpha_2_tip_deg = numpy.rad2deg(alpha_2_tip_rad)

        alpha_3_tip_rad = numpy.arctan((r_m_pointer/r_tip_pointer) *numpy.tan(numpy.deg2rad(alpha_3_pointer)))
        alpha_3_tip_deg = numpy.rad2deg(alpha_3_tip_rad)

        beta_2_tip_rad = numpy.arctan((r_m_pointer/r_tip_pointer) *numpy.tan(numpy.deg2rad(alpha_2_pointer)) - (r_tip_pointer/r_m_pointer)*
        beta_2_tip_deg = numpy.rad2deg(beta_2_tip_rad)

        beta_3_tip_rad = numpy.arctan((r_m_pointer/r_tip_pointer) *numpy.tan(numpy.deg2rad(alpha_3_pointer)) + (r_tip_pointer/r_m_pointer)*
        beta_3_tip_deg = numpy.rad2deg(beta_3_tip_rad)

        U_tip = U_pointer * (r_tip_pointer / r_m_pointer)
        V_2_tip = C_a_2/np.cos(beta_2_tip_rad)
        C_2_tip = C_a_2/np.cos(alpha_2_tip_rad)

        C_2_tip = C_a_2/np.cos(alpha_2_tip_rad)
        C_3_tip = C_a_3/np.cos(alpha_3_tip_rad)
        V_3_tip = C_a_3/np.cos(beta_3_tip_rad)

        T_2_tip = T_02 - (C_2_tip**2)/(2*c_p_gas*1000)
        T_3_tip = T_03 - (C_3_tip**2)/(2*c_p_gas*1000)


        a_2_tip = np.sqrt(gamma_g*T_2_tip*R*1000)
        a_3_tip = np.sqrt(gamma_g*T_3_tip*R*1000)

        M_2_rel_tip = V_2_tip / a_2_tip
        M_2_tip = C_2_tip / a_2_tip
        M_3_rel_tip = V_3_tip / a_3_tip

        M_2_rel_tip = V_2_tip / a_2
        M_2_tip = C_2_tip / a_2


        return alpha_2_tip_deg, alpha_3_tip_deg, beta_2_tip_deg, beta_3_tip_deg, U_tip, V_2_tip, C_2_tip, M_2_rel_tip, M_2_tip, M_3_rel_tip


    def calc_tip_hub_reaction(c_a_3_pointer, c_a_2_pointer ,beta_2_hub, beta_2_tip, beta_3_hub, beta_3_tip, U_hub, U_tip):

        reaction_hub = (c_a_3_pointer * numpy.tan(numpy.deg2rad(beta_3_hub)) - c_a_2_pointer*numpy.tan(numpy.deg2rad(beta_2_hub)))/(2*U_hub
        reaction_tip = (c_a_3_pointer * numpy.tan(numpy.deg2rad(beta_3_tip)) - c_a_2_pointer*numpy.tan(numpy.deg2rad(beta_2_tip)))/(2*U_tip
        return reaction_hub, reaction_tip


class aerostructural():
    def calc_structural(an_squared_pointer, area_2_pointer, U_meanline_pointer):
        N = numpy.sqrt((an_squared_pointer)/area_2_pointer)
        omega = N*2*numpy.pi/60
        r_meanline = U_meanline_pointer/omega
        h = (area_2_pointer * (N/60))/U_meanline_pointer
        r_hub = r_meanline - (h/2)
        r_tip = r_meanline + (h/2)
        return N, omega, r_hub, r_tip, r_meanline, h
```

```python
class aerodynamic_losses():
    """
    Profile losses calculated in this class.
    """
    class profile_losses():
        """
        Profile losses calculated in this class.
        """
        def figure_2_3a(pitch_chord_ratio, exit_flow_angle):
            fig_2_3a = pd.read_csv(r'_input_database\figure_2_3a.csv')
            X = fig_2_3a[['pitch_chord_ratio', 'exit_flow_angle']]
            y = fig_2_3a['K_P_1']

            X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

            model = LinearRegression()
            model.fit(X_train, y_train)

            X = pd.DataFrame({'pitch_chord_ratio': [pitch_chord_ratio], 'exit_flow_angle': [exit_flow_angle]})
            K_P_1 = model.predict(X)

            return K_P_1[0]

        def figure_2_3b(pitch_chord_ratio, exit_flow_angle):
            fig_2_3b = pd.read_csv(r'_input_database\figure_2_3b.csv')
            X = fig_2_3b[['pitch_chord_ratio', 'exit_flow_angle']]
            y = fig_2_3b['K_P_2']

            X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

            model = LinearRegression()
            model.fit(X_train, y_train)

            X = pd.DataFrame({'pitch_chord_ratio': [pitch_chord_ratio], 'exit_flow_angle': [exit_flow_angle]})
            K_P_2 = model.predict(X)

            return K_P_2[0]

        def figure_2_4(beta_b1, beta_b2):
            beta_eff = beta_b1 + beta_b2
            fig_2_4 = pd.read_csv(r'_input_database\figure_2_4.csv')
            X = fig_2_4[['beta_b1_b2']]
            y = fig_2_4['tmax_and_c']

            X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

            model = LinearRegression()
            model.fit(X_train, y_train)

            X = pd.DataFrame({'beta_b1_b2': [beta_eff]})
            tmax_and_c = model.predict(X)

            return tmax_and_c[0]

        def figure_2_5(beta_b1, beta_b2):
            fig_2_5 = pd.read_csv(r'_input_database\figure_2_5.csv')
            X = fig_2_5[['beta_b1', 'beta_b2']]
            y = fig_2_5['Stagger Angle']

            X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

            model = LinearRegression()
            model.fit(X_train, y_train)

            X = pd.DataFrame({'beta_b1': [beta_b1], 'beta_b2': [beta_b2]})
            stagger_angle = model.predict(X)

            return stagger_angle[0]


        def figure_2_6(x_value, x=True):
            """
            x = True for Rotor
            """
            fig_2_6 = pd.read_csv(r'_input_database\figure_2_6.csv')
            rhrt = fig_2_6['x'].values
            y1 = fig_2_6['Rotor'].values
            y2 = fig_2_6['Nozzle'].values

            if x == True:
                interp_func = interp1d(rhrt, y1, kind='cubic')
                interpolated_y = interp_func(x_value)
            else:
                interp_func = interp1d(rhrt, y2, kind='cubic')
                interpolated_y = interp_func(x_value)

            return interpolated_y

        def figure_2_7(x_value):
```

```python
        fig_2_7 = pd.read_csv(r'_input_database\figure_2_7.csv')
        x = fig_2_7['M_1_hub'].values
        y = fig_2_7['delta_p_q_1_hub'].values

        interp_func = interp1d(x, y, kind='cubic')
        interpolated_y = interp_func(x_value)

        return interpolated_y


    """
    The following section give information about the symbols to be used for the following class.
    This is established from the information provided in Axial and Radial Turbines Part A - Moustapha.
    """
    def calc_K_p(M_1, M_2, M_1_rel_hub, P_1, P_2 ,r_tip, r_hub, beta_in, beta_out, zweifel_pointer):
        """
        Using corrected effects of exit mach number.
        Function used to determine K_accel.
        Equation (2.7), (2.8)
        IMPORTANT:
        FOR STATOR
        beta_in -> alpha_1
        beta_out -> alpha_2
        FOR ROTOR
        beta_in -> beta_2
        beta_out -> beta_3

        """
        #  ============= K_p_star =============
        # get the stagger angle from figure 2.5
        stagger_angle = aerodynamic_losses.profile_losses.figure_2_5(beta_in, beta_out)      # [deg]
        tmax_and_c = aerodynamic_losses.profile_losses.figure_2_4(beta_in, beta_out)

        pitch_chord_ratio = (zweifel_pointer / (2 * (numpy.tan(numpy.radians(beta_in)) + numpy.tan(numpy.radians(beta_out)) ) * (numpy.
        pitch_axial_chord_ratio = pitch_chord_ratio/numpy.cos(numpy.radians(stagger_angle))
        K_P_2 = aerodynamic_losses.profile_losses.figure_2_3b(pitch_chord_ratio, beta_out)         # beta_3 in other code
        K_P_1 = aerodynamic_losses.profile_losses.figure_2_3a(pitch_chord_ratio, beta_out)         # beta_3 in other code
        K_p_star = (K_P_1 + (abs(beta_in / beta_out) * (beta_in / beta_out))*(K_P_2 - K_P_1)) * ((tmax_and_c / 0.2)**(beta_in / beta_ou
        #  =============== K_sh ===============
        k = 1.333333

        if M_1_rel_hub <= 0.4:
            A = 0
        if M_1_rel_hub > 0.4:
            A = 0.75 * ((M_1_rel_hub - 0.4)**(7/4))

        B = A * (r_hub/r_tip)
        C = 1 - ((1 + ((k - 1) / (2)) * M_1**2)**(k/(k - 1)))
        D = 1 - ((1 + ((k - 1) / (2)) * M_2**2)**(k/(k - 1)))

        K_sh = B * (P_1 / P_2) * (C / D)

        #  ============== K_accel ==============
        if M_2 <= 0.2:
            K_1 = 1.0
        if M_2 > 0.2:
            K_1 = 1 - 1.25 * (M_2 - 0.2)
        K_2 = (M_1/M_2)**2
        K_accel = 1 - K_2 * (1 - K_1)

        #  ================ K_p ================
        K_p = 0.914 * ((2/3) * K_p_star * K_accel + K_sh)

        return K_p, pitch_chord_ratio, K_accel, stagger_angle,  pitch_chord_ratio, pitch_axial_chord_ratio, K_1, K_2

class secondary_losses():

    def calc_K_s(K_accel, AR, beta_in, beta_out):
        """
        FOR STATOR
        beta_in -> alpha_1
        beta_out -> alpha_2
        FOR ROTOR
        beta_in -> beta_2
        beta_out -> beta_3
        """

        if AR <= 2:
            f_AS = (1- 0.25 * np.sqrt(2 - AR))/(AR)
        if AR > 2:
            f_AS = 1/AR
        alpha_m = np.arctan(0.5 * (np.tan(np.radians(beta_in)) - np.tan(np.radians(beta_out))))
        A = np.cos(np.radians(beta_out))/np.cos(np.radians(beta_in))
        B = 2 * (np.tan(np.radians(beta_in)) + np.tan(np.radians(beta_out))) * np.cos(alpha_m)
        C = ((np.cos(np.radians(beta_out)))**2) / ((np.cos(alpha_m))**3)
        K_s_star = 0.0334 * f_AS * A *  (B**2) * C
        K_3 = 1 / ((AR)**2)
        K_cs = 1 - K_3 * (1 - K_accel)
        K_s = 1.2 * K_s_star * K_cs
```

```
        return K_s

class trailing_edge_losses_rotor():

    def figure_2_10(x_value, beta_in, beta_out):
        """
        f -> delta_phi_squared_TE
        FOR STATOR
        beta_in -> alpha_1
        beta_out -> alpha_2
        FOR ROTOR
        beta_in -> beta_2
        beta_out -> beta_3
        """
        fig_2_10 = pd.read_csv(r'_input_database\figure_2_10.csv')
        r_te_o = fig_2_10['r_te_o'].values
        impulse_blading = fig_2_10['impulse_blading'].values
        stator_vanes = fig_2_10['stator_vanes'].values

        interp_func_1 = interp1d(r_te_o, impulse_blading, kind='cubic')
        interp_func_2 = interp1d(r_te_o, stator_vanes, kind='cubic')
        f_alpha  = interp_func_1(x_value)
        f_zero = interp_func_2(x_value)

        f = f_zero + (abs(beta_in / beta_out) * (beta_in / beta_out))*(f_alpha - f_zero)

        return f


    def required_vals(h, stagger_angle, r_meanline, pitch_axial_chord_ratio, beta_3):
        c_true = (h)/AR_rotor
        c_a = (h * np.cos(np.radians(stagger_angle)))/AR_rotor
        N = math.floor((2 * np.pi * r_meanline) /(pitch_axial_chord_ratio * c_a))
        o = (pitch_axial_chord_ratio * c_a) * np.cos(np.radians(beta_3))

        return c_true, c_a, N, o


    def K_TET(M_2, beta_in, beta_out, h, stagger_angle, r_meanline, pitch_axial_chord_ratio):
        """
        M_2 -> Use relative M_2_rel for the rotor.
        FOR STATOR
        beta_in -> alpha_1
        beta_out -> alpha_2
        FOR ROTOR
        beta_in -> beta_2
        beta_out -> beta_3
        """
        c_true, c_a, N, o = aerodynamic_losses.trailing_edge_losses_rotor.required_vals(h, stagger_angle, r_meanline, pitch_axial_chord_


        r_to_o = TE_rotor/o
        f = aerodynamic_losses.trailing_edge_losses_rotor.figure_2_10(r_to_o ,beta_in, beta_out)

        term1 = ((gamma_g - 1) / 2) * M_2**2
        term2 = (1 / (1 - f)) - 1
        numerator = ((1 - term1 * (term2))**(-gamma_g / (gamma_g - 1)) )- 1
        denominator = 1 - ((1 + term1))**(-gamma_g / (gamma_g - 1))
        K_TE = numerator / denominator
        throat_opening = o

        return K_TE, N, c_true, c_a, throat_opening

class trailing_edge_losses_stator():

    def figure_2_10(x_value, beta_in, beta_out):
        """
        f -> delta_phi_squared_TE
        FOR STATOR
        beta_in -> alpha_1
        beta_out -> alpha_2
        FOR ROTOR
        beta_in -> beta_2
        beta_out -> beta_3
        """
        fig_2_10 = pd.read_csv(r'_input_database\figure_2_10.csv')
        r_te_o = fig_2_10['r_te_o'].values
        impulse_blading = fig_2_10['impulse_blading'].values
        stator_vanes = fig_2_10['stator_vanes'].values

        interp_func_1 = interp1d(r_te_o, impulse_blading, kind='cubic')
        interp_func_2 = interp1d(r_te_o, stator_vanes, kind='cubic')
        f_alpha  = interp_func_1(x_value)
        f_zero = interp_func_2(x_value)

        f = f_zero + (abs(beta_in / beta_out) * (beta_in / beta_out))*(f_alpha - f_zero)

        return f


    def required_vals(h, stagger_angle, r_meanline, pitch_axial_chord_ratio, beta_3):
```

```python
        c_true = (h)/AR_vane
        c_a = (h * np.cos(np.radians(stagger_angle)))/AR_vane
        N = math.floor((2 * np.pi * r_meanline) /(pitch_axial_chord_ratio * c_a))
        o = (pitch_axial_chord_ratio * c_a) * np.cos(np.radians(beta_3))

        return c_true, c_a, N, o


    def K_TET(M_2, beta_in, beta_out, h, stagger_angle, r_meanline, pitch_axial_chord_ratio):
        """
        M_2 -> Use relative M_2_rel for the rotor.
        FOR STATOR
        beta_in -> alpha_1
        beta_out -> alpha_2
        FOR ROTOR
        beta_in -> beta_2
        beta_out -> beta_3
        """
        c_true, c_a, N, o = aerodynamic_losses.trailing_edge_losses_stator.required_vals(h, stagger_angle, r_meanline, pitch_axial_chor


        r_to_o = TE_vane/o
        f = aerodynamic_losses.trailing_edge_losses_stator.figure_2_10(r_to_o ,beta_in, beta_out)

        term1 = ((gamma_g - 1) / 2) * M_2**2
        term2 = (1 / (1 - f)) - 1
        numerator = ((1 - term1 * (term2))**(-gamma_g / (gamma_g - 1)) )- 1
        denominator = 1 - ((1 + term1))**(-gamma_g / (gamma_g - 1))
        K_TE = numerator / denominator
        throat_opening = o
        return K_TE, N, c_true, c_a, throat_opening


def efficiency_calculations(K_stator, K_rotor, M_2, M_3_rel, C_2, V_3):
    zeta_N = K_stator/(1 + 0.5 * gamma_g * M_2**2)
    zeta_R = K_rotor/(1 + 0.5 * gamma_g * M_3_rel**2)
    eta_tt = 1 / (1 + ((zeta_N * C_2**2 + zeta_R * V_3**2) / (2 * c_p_gas * 1000 * (T_01 - T_03))))
    eta_tt = eta_tt * 100
    return eta_tt

def efficiency_final(eta_tt, h, beta_3, r_tip, r_meanline):
    delta_n = 0.93 * (eta_tt/100) * ((tip_clearance/100)/(h * np.cos(np.radians(beta_3)))) * (r_tip/r_meanline)
    eta_final = eta_tt - delta_n
    return delta_n, eta_final

def losses_off_design(K_p_rotor, K_s_rotor, K_stator, K_1_rotor, K_2_rotor, pitch_chord_ratio_rotor, pitch_axial_chord_ratio_rotor, c_t
    # Primary
    phi_squared_P0 = 1 / (1 + ((K_p_rotor) / (K_1_rotor + K_2_rotor * K_p_rotor)))
    s = pitch_chord_ratio_rotor * c_true
    d_s = LE_diameter_rotor/s
    graph_x = (d_s)**(-1.6) * (np.cos(np.radians(beta_2)) / np.cos(np.radians(beta_3)))**(-2) * (np.radians(incidence))

    def figure_2_34(graph_x):
        fig_2_34 = pd.read_csv(r'_input_database\figure_2_34.csv')
        x = fig_2_34['graph_x'].values
        y = fig_2_34['deg_of_accel'].values

        interp_func = interp1d(x, y, kind='cubic')
        interpolated_y = interp_func(graph_x)

        return interpolated_y

    deg_of_accel = figure_2_34(graph_x)
    phi_squared_P = phi_squared_P0 - deg_of_accel
    K_p_od = (K_1_rotor * (1-phi_squared_P)) / (phi_squared_P - K_2_rotor * (1-phi_squared_P))


    # Secondary
    d_c = LE_diameter_rotor/c_true
    graph_x_2 = (d_c)**(-0.3) * (np.cos(np.radians(beta_2)) / np.cos(np.radians(beta_3)))**(-1.5) * ((np.radians(incidence))/(np.radian
    if graph_x_2 < 0.27:
        def figure_2_35(graph_x_2):
            fig_2_35 = pd.read_csv(r'_input_database\figure_2_35.csv')
            x = fig_2_35['x'].values
            y = fig_2_35['K_K_des'].values

            interp_func = interp1d(x, y, kind='cubic')
            interpolated_y = interp_func(graph_x_2)

            return interpolated_y

        K_K_des = figure_2_35(graph_x_2)
        K_s_od =  K_K_des * K_s_rotor

        # Trailing Edge TO BE CONFIRMED
        K_TET_od, N_rotor_od, c_true_rotor_od, c_a_rotor_od,dummy = aerodynamic_losses.trailing_edge_losses_rotor.K_TET(M_2_rel_od, bet

        K_rotor_od = K_p_od + K_s_od + K_TET_od

        eta_tt_od = aerodynamic_losses.efficiency_calculations(K_stator, K_rotor_od, M_2_rel_od, M_3_rel_od, C_2, V_3)
        delta_n_od, eta_final_od = aerodynamic_losses.efficiency_final(eta_tt_od, h, beta_3, r_tip, r_meanline)
```

```python
        else:
            K_K_des, K_s_od, eta_tt_od, delta_n_od, eta_final_od = 0,0,0,0,0


        return eta_tt_od, delta_n_od, eta_final_od


class off_design():
    def calc_off_design(A_3, U_mean,beta_3,Ca_2, Cw_2, beta_2, a_2, a_3):
        U_mean_od = U_mean *0.9
        C_w_2_od = Cw_2
        flow_coeff_2_od = Ca_2/U_mean_od
        V_w_2_od = Cw_2 - U_mean_od
        alpha_2_rel_od = np.arctan(V_w_2_od/Ca_2)
        alpha_2_rel_od_deg = np.rad2deg(alpha_2_rel_od)
        incidence_2 = alpha_2_rel_od_deg - beta_2 #beta_2 is the blade angle -> alpha_2_rel from previous calculations
        v_2_od = np.sqrt(V_w_2_od**2 + Ca_2**2) #relative velocity on the hypoteneuse (total relative velocity at 2)
        M_2_rel_od = v_2_od / a_2 #assumed speed of sound at 2 OD = speed of sound on design.
        T_2 = a_2**2/(gamma_g*R*1000)

        LHS = R*m_dot_3/A_3
        error_threshold = LHS * 0.01 #1 percent error of the LHS

        Ca_3_range = np.linspace(100,400,1000)
        Ca_3_od = 0

        for i in Ca_3_range:
            V_w_3_od = i * np.tan(np.deg2rad(beta_3))
            C_w_3_od = V_w_3_od - U_mean_od
            C_3_od = np.sqrt(i**2 + C_w_3_od**2)
            T_3_od = T_03 - (C_3_od**2)/(2*1000*c_p_gas)
            P_3_od = P_03 * (T_3_od/T_03)**(gamma_g/(gamma_g-1))
            RHS = i *P_3_od/T_3_od
            a_3_od = math.sqrt(gamma_g*R*1000*T_3_od)

            if np.abs(LHS - RHS) < error_threshold:
                Ca_3_od = i
                alpha_3_od = np.rad2deg(np.arctan(C_w_3_od/Ca_3_od))
                rho_3_od = P_3_od/(R*T_3_od)
                work_od_cw = U_mean_od*(C_w_3_od + C_w_2_od)
                work_od_vw = U_mean_od*(V_w_3_od + V_w_2_od)
                flow_coeff_3_od = Ca_3_od/U_mean_od
                V_3_od = np.sqrt(V_w_3_od**2 + Ca_3_od**2)
                M_3_rel_od = V_3_od/a_3_od
                # For physics check
                P_2 = P_3_od * ((T_2/T_3_od) ** (gamma_g/(gamma_g - 1)))
                P_02_rel = P_2*(1+ (gamma_g-1)/2 * M_2_rel_od**2)**(gamma_g/(gamma_g-1))
                P_03_rel = P_3_od*(1+ (gamma_g-1)/2 * M_3_rel_od**2)**(gamma_g/(gamma_g-1))
                break

        if Ca_3_od == 0: #if nothing happens, return everything as zero
            V_w_3_od =0
            C_w_3_od = 0
            C_3_od = 0
            T_3_od = 0
            P_3_od = 0
            Ca_3_od = 0
            alpha_3_od = 0
            rho_3_od = 0
            work_od_cw = 0
            work_od_vw = 0
            flow_coeff_3_od = 0
            M_3_rel_od= 0
            P_02_rel = 0
            P_03_rel = 0

            return T_3_od, rho_3_od, P_3_od, alpha_3_od, alpha_2_rel_od_deg, flow_coeff_2_od, incidence_2, v_2_od, C_w_3_od,Ca_3_od,U_mean_

        else:
            return T_3_od, rho_3_od, P_3_od, alpha_3_od, alpha_2_rel_od_deg, flow_coeff_2_od, incidence_2, v_2_od, C_w_3_od,Ca_3_od,U_mean_



def verify_zweifel_rotor(r_hub, h, N,c, alpha_1, alpha_2):
    s = (2*np.pi * (r_hub + 0.5*h))/N
    zxr = 2 * ((s/c)) * (np.tan(np.radians(alpha_1)) + np.tan(np.radians(alpha_2))) * np.cos(np.radians(alpha_2))**2
    return zxr

def verify_zweifel_stator(r_hub, h, N,c, alpha_1, alpha_2):
    s = (2*np.pi * (r_hub + 0.5*h))/N
    zxs = 2 * ((s/c)) * (np.tan(np.radians(alpha_1)) + np.tan(np.radians(alpha_2))) * np.cos(np.radians(alpha_2))**2
    return zxs
```